HackOSsim

Simone Sampognaro s322918 , Manuel Firrera s329226
Alessandro Amoretti s330933 , Alessandro Mulassano s330263

Prof. Stefano Di Carlo
01GYKUV - Computer architectures and operating systems

February 19, 2024

Contents

1 Installation guide

1.1 Installing QEMU

QEMU, which stands for Quick Emulator, is an open-source virtualization software that allows you to emulate the hardware of various architectures, such as x86, ARM, PowerPC, and others. It provides a platform for running virtual machines (VMs) on a host system. QEMU can emulate the entire system, including the processor, memory, storage devices, and peripheral devices, making it a versatile tool for virtualization and emulation.

1.1.1 Installing QEMU on Windows

1. Download and install QEMU, there is a separate download page for pre-built QEMU Windows executables.
2. Add QEMU installation path to your PATH enviroment variable
3. Verify installation by typing on command prompt:

```
qemu-img --version
```

1.1.2 Installing QEMU on Linux

Open a linux terminal and run the following command line:

```
sudo apt-get install qemu-system
```

1.2 Downloading FreeRTOS

FreeRTOS, or Free Real-Time Operating System, is an open-source real-time operating system (RTOS) designed for embedded systems and microcontrollers. It provides a small, efficient, and portable kernel that is suitable for a wide range of applications where real-time performance is crucial.
Download it from `https://www.freertos.org/index.html`

1.3 Installing other useful tools on Windows

- Arm GNU toolchain - `https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads`
- CMake - `https://cmake.org/download/`
- Make - `https://www.technewstoday.com/install-and-use-make-in-windows/`

Add them to installation path to your PATH enviroment variable and then verify installation by typing on command prompt: tool –version

1.4 Installing other useful tools on Linux

1. Download the arm-gcc-none-eabi compiler from the following link: `https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads`
2. Install the compiler (in linux installation consists in adding the compiler bins directory to PATH)
   - Move to the arm-gcc-none-eabi directory and unzip it
   - Run the following command line(make sure you still are in the arm-gcc-none-eabi directory):

```
install_dir=$(pwd)
```

- Update PATH running the following command line (repeat b. and c. every time you open a new terminal):

```
\$PATH: \$\{install\_dir\}/gcc-arm-none-eabi-10.3-2021.10/bin
```

- Verify correct installation by typing the following command:

```
arm-none-eabi-gcc --version
```

2 FreeRTOS demo application

The RTOS source code download includes a pre-configured demonstration project for each RTOS port. More informations can be found on the FreeRTOS Demos Applications documentation page. We will refer to a FreeRTOS kernel demo that targets the Arm Cortex-M3 mps2-an385 QEMU model. (`https://www.freertos.org/freertos-on-qemu-mps2-an385-model.html`)

2.1 Building and exececuting the demo

1. Open FreeRTOS/Demo/CORTEX_MPS2_QEMU_IAR_GCC/main.c, and set mainCREATE_SIMPLE_BLINKY_DEMO_ONLY to generate either the simply blinky demo, or the comprehensive test and demo application, as required.

   (a) Simple "blinky" demo configuration: they are contained in a single source file and implement a subset of the functionality described on the hardware independent demo functions page. It's the default configuration in which the constant is set to 1.

   (b) Comprehensive test/demo configuration: they projects create all or a subset of the "common demo tasks" - so called because the tasks are common to all comprehensive demos. The number of tasks created depends on the resources available on the target hardware (microcontroller or microprocessor) - the amount of RAM available for task stacks being the limiting factor. In order to get this configuration you have to change the constant to a number different from 1.

   In this section we leave the default configuration.

2. Open a command prompt and go to the FreeRTOS/Demo/CORTEX_MPS2_QEMU_IAR_GCC/build/gcc directory.

3. Type "make" in the command prompt. The project should build without any compiler errors or warnings. Hint: Use the "-j" parameter to speed the compilation by using more cores on your host computer. For example, if you have four cores available you can build four C files at once by entering "make -j4". A successful build creates the elf file FreeRTOS/Demo/CORTEX_MPS2_QEMU_IAR_GCC/build/gcc/output/RTOSDemo.out.

4. Start QEMU with the following command line, replacing [path-to] with the correct path to the RTOSDemo.out file generated by the GCC build:

```
qemu-system-arm -machine mps2-an385 -cpu cortex-m3 -kernel [path-to]/
    RTOSDemo.out
```

5. If everything works correctly a window of QEMU will be opened and going on the section "Visualize" and choosing the parameter "serial()" the window will show the program running.

3 Exercises

In order to understand how to approach to FreeRTOS and its APIs, each of the component of the group focused on some of the provided functionalities, managing to develop several demos, which can be found on the Git repository with proper explanations, to show how we can use FreeRTOS.
These are the developed demos:

1. Priorities: three tasks with ascending priority levels are scheduled by priority, in order to give a sorted execution.
2. RoundRobin: two tasks share the same priority level, the default scheduler will work in a Round Roubin fashion.
3. Message Buffer: two tasks communicating with each other by exchanging messages with the messageBuffer.
4. Queue: three tasks try to fill a queue with a length of 2, so the one with lower priority cannot get into the queue.
5. Timer-Interrupt Service Routine (T-ISR): A software timer-interrupt is invoked to release a semaphore needed by a task.
6. Events: A high priority task waits on a bit of the event group, thus leaving the CPU to lower priority tasks. The event is set by a periodic timer.
7. Mutex: using mutexes to control access to a resource that is shared among three tasks, a global variable.
8. Semaphores: using binary semaphores to control the order of execution of three tasks.
9. Notifications: a simulated peripheral sends periodic notifications to a task handler, which then notifies one or more tasks based on the event variable's value, ultimately unblocking them. Subsequently, the tasks are allowed to read and print the notification value on the standard output.

4 Customizations

In this section we cover the customizations implemented to the kernel.

4.1 Heap memory management

FreeRTOS dynamically allocates memory in the heap space through the use of the vPortMalloc function, which is a lightweight and deterministic implementation of the ANSI C malloc function. Our modifications consists in two new implementations of the memory allocation.

4.1.1 Heap_6

This scheme uses a worst fit algorithm with coalescence, which is capable of joining two adjacent free blocks into a single bigger one. This can be simply implemented in a FreeRTOS instance by putting the C file into the MemMang folder and modifying the Makefile accordingly. Our custom allocator works by keeping track of the free blocks by inserting them into an address-sorted linked list. Every time the vPortMalloc function is called, the biggest free block is selected, updated by removing the required space for the allocation and the remaining block is reinserted into the list. Sorting the blocks by address comes in hand when dealing with the coalescence problem. The algorithm checks if the previous block is free and adjacent to the newly inserted block, joining them if true; the same can be said for the subsequent one. This mechanism presents some advantages (green) and disadvantages (red), here's an overall of the considerations due to the assessments which followed the implementation:

- Simple implementation: The logic is relatively straightforward, involving only finding the largest memory block instead of searching for a block of specific size.
- Better suited for certain workloads: In scenarios where large memory blocks are frequently requested, Worst Fit can be advantageous as it is more likely to have larger blocks available, leading to faster allocations without the need for memory compaction.
- External fragmentation: It may lead to higher external fragmentation as free memory blocks can be smaller and harder to efficiently utilize.
- Poor memory utilization: Since it allocates the largest available block, it can lead to more memory wastage compared to more efficient methods like First Fit.

4.1.2 Heap_7

Heap_7 is a buddy system memory allocation method. The buddy system ensures that memory is efficiently used and helps prevent fragmentation. When a memory block is released, the system checks if its buddy (the adjacent block of the same size) is also free. If so, the two buddy blocks are merged back into a larger block, maintaining a power-of-two size. To achieve the desired goal, the algorithm has to follow some steps:

- Calculate the nearest power-of-two of the requested allocation size.
- Search for a block of adequate size, using a best-fit approach.
- If not present, divide a bigger block into smaller ones to accommodate the new data.

This mechanism presents some advantages (green) and disadvantages (red), here's an overall of the considerations due to the assessments which followed the implementation:

- Reduced fragmentation: Since it allocates memory blocks that are exactly the required size, it tends to reduce both internal and external fragmentation.
- Good memory utilization: When the system works well, it can ensure efficient memory usage without significant wastage.
- Efficient implementation: Its binary division-based structure makes the allocation and deallocation of memory relatively efficient.
- Implementation complexity: The Buddy System requires a more complex implementation compared to First Fit due to the need to manage memory blocks hierarchically.
- Management overhead: It can lead to a higher management overhead compared to First Fit, especially in terms of extra space required to maintain information about buddy blocks.

4.2 Scheduler

In this section we discuss the customization of the scheduler consisting in the implementation of the Aging mechanism using a priority based scheduling with time-slicing.
The FreeRTOS scheduler doesn't provide by default a mechanism, other than priority inheritance, to manage some of the problems which could occur during tasks scheduling. When the number of tasks and the workloads increase, the system could end up in a saturated state in which it wouldn't be possible to grant full execution to all the tasks. In order to respond to these inconveniences we decided to implement the Aging mechanism, which is responsible to "enhance" the priorities of those tasks that didn't manage to get control over the CPU for a certain period of time, thus needing a priority increase to have better chances to finally execute. This treatment allows each task to reach eventually their execution phase and can be furtherly improved by adding a "Reset Priority" option that permits to reset the tasks' priority after having received a priority increase which led to enter the running state.
This mechanism presents some advantages (green) and disadvantages (red), here's an overall of the considerations due to the assessments which followed the implementation:

- Improved responsiveness: Aging mechanisms ensure that tasks that have been waiting for longer periods are given higher priority, allowing them to execute sooner. This can lead to better responsiveness, reduced latency, and improved overall system performance, especially in systems with a mix of short- and long-running tasks.
- Fairness: By adjusting task priorities based on their execution history, aging mechanisms can provide fairness in task scheduling. Tasks that have been waiting for longer periods are given higher priority, ensuring that all tasks have an opportunity to execute and preventing any single task from monopolizing system resources indefinitely.
- Dynamic adaptation to workload changes: Aging mechanisms allow the operating system to dynamically adapt task priorities based on runtime conditions and workload changes. Tasks that were previously low priority but have become more critical due to changing circumstances can have their priorities adjusted accordingly, ensuring that system resources are allocated optimally.
- Performance overhead: The overhead of tracking and updating task priorities based on their execution history can impact system performance, especially in resource-constrained embedded systems with limited processing power and memory. The additional processing required for aging can potentially increase context switching overhead and degrade overall system responsiveness.
- Difficulty in tuning parameters: Aging mechanisms often rely on configurable parameters such as aging rates or thresholds to determine how task priorities should be adjusted. Finding the optimal values for these parameters can be challenging and may require careful tuning and testing to ensure optimal system performance and stability across different workload scenarios.
- Increased resource usage: The data structures and algorithms required to implement aging mechanisms consume additional system resources, including memory and processing power. In resource-constrained embedded systems, the overhead of managing aging-related data structures may become significant and could impact the overall system's ability to meet real-time deadlines.

In conclusion, while aging mechanisms can offer benefits such as improved responsiveness and fairness in task scheduling, they also come with trade-offs in terms of complexity, performance overhead, and potential challenges in parameter tuning and system resource usage. It is crucial to understand what are the system's purposes and its usage in order to properly state if an implementation of this kind of mechanism could actually be beneficial and be able to bring the desired improvements.

5 Statements

The workload has been divided in the following way: the first part has been done all together, since everybody needed to understand properly how to install and use FreeRTOS and QEMU, both on Windows and Linux. For the second point, each member realized some exercises in order to get accustomed to different functionalities, in particular messageBuffer and Queue have been made by Mulassano, Priorities and T-ISR have been made by Amoretti, RoundRobin and Events have been made by Firrera and Mutex, Semaphores and Notifications have been made by Sampognaro. Regarding kernel customizations we split the workload in two groups: Firrera and Mulassano implemented the customs related to memory management, whereas Amoretti and Sampognaro customized the scheduler.